

The Second Project Report

Samira Vaez Barenji

Student ID Number: 9921384

In this project, you can find a dynamic programming solution to the word wrapping problem. The Brute force solution for this problem doesn't give optimal solution in all cases. So, using dynamic programming and storing the results of subproblems, we minimize the total cost of lines resulting from breaking words into a few lines. The procedure of the algorithm is as follows:

First we compute costs of all possible lines in a 2D matrix *line_cost[][]*. The value *line_cost[i][j]* indicates the cost to put words from *i* to *j* in a single line where *i* and *j* are indexes of words in the input sequences. If a sequence of words from *i* to *j* cannot fit in a single line, then *line_cost[i][j]* is considered infinite (to avoid it from being a part of the solution). Once we have the *line_cost[][]* matrix constructed, we can calculate total cost.

This problem has overlapping subproblem property. So Dynamic Programming is used to store the results of subproblems. The array *total_cost[]* can be computed from left to right, since each value depends only on earlier values.

To print the output, we keep track of what words go on what lines, we can keep a parallel *s* array that points to where each *total_cost* value came from. The last line starts at word *s[n]* and goes through word *n*. The previous line starts at word *s[s[n]]* and goes through word *s[n] - 1*, etc. The function *print_result()* uses returned *s[]* array to print the solution.

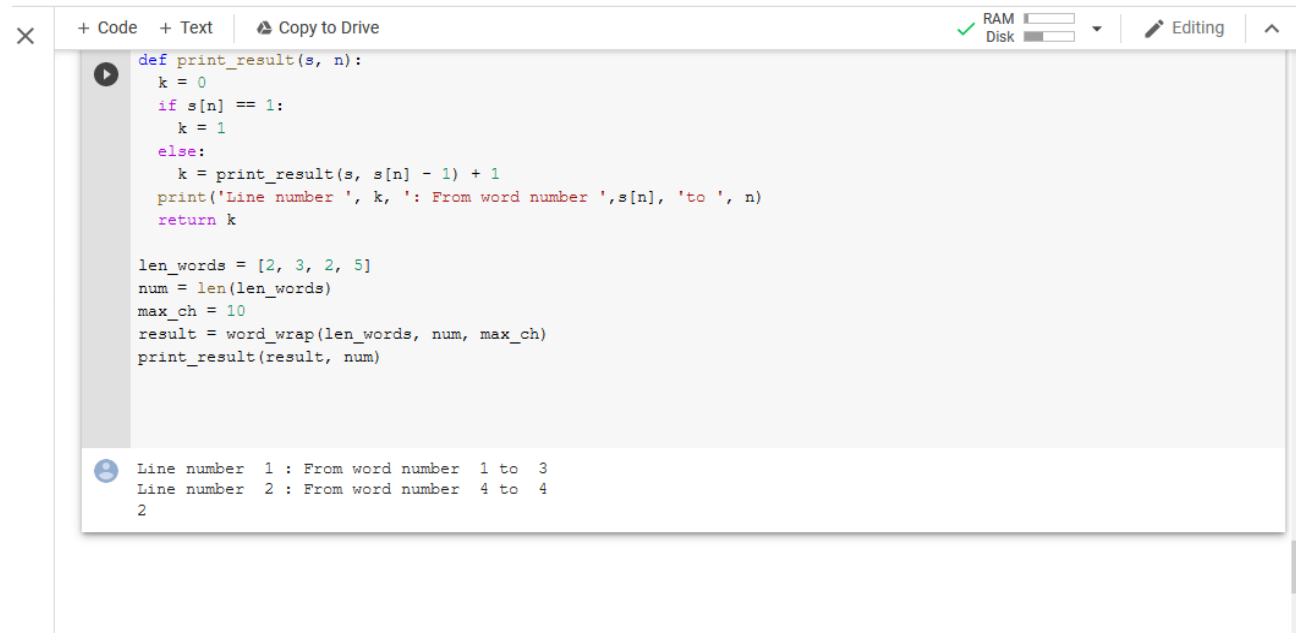
The *word_wrap* function, takes three parameters which are as follows:

- a) an array *len_of_words[]* that represents lengths of words in a sequence. The value *len_of_words[i]* indicates length of the *i*th word (*i* starts from 1) in the input sequence.
- b) A variable called *n* which is the length of *len_of_words[]*.

- c) A variable called *max_char* which is the maximum number of characters that can fit in a line (line width).

You can change the value of each of these parameters to your liking. In the figure below you can see a picture of the output of the algorithm for line width 10 and *len_of_words* = [2, 3, 2, 5].

In this algorithm, since our outer loop runs from 1 to n and our inner loop runs from i to n , we have a polynomial time complexity of $O(n^2)$ and our *space* array is a 2D array which is of size $N*N$ and gives us polynomial space complexity of $O(N^2)$.



The screenshot shows a code editor with a toolbar at the top containing a close button, '+ Code', '+ Text', 'Copy to Drive', RAM and Disk usage indicators, and an 'Editing' mode button. The code defines a recursive function `print_result(s, n)` that prints the line number and the range of words it contains. It then initializes `len_words = [2, 3, 2, 5]`, calculates the number of words, sets `max_ch = 10`, and calls `word_wrap` and `print_result`. The output at the bottom shows two lines: 'Line number 1 : From word number 1 to 3' and 'Line number 2 : From word number 4 to 4'.

```
def print_result(s, n):
    k = 0
    if s[n] == 1:
        k = 1
    else:
        k = print_result(s, s[n] - 1) + 1
    print('Line number ', k, ': From word number ', s[n], 'to ', n)
    return k

len_words = [2, 3, 2, 5]
num = len(len_words)
max_ch = 10
result = word_wrap(len_words, num, max_ch)
print_result(result, num)
```

Line number 1 : From word number 1 to 3
Line number 2 : From word number 4 to 4